ISO/IEC JTC 1/SC 22/WG23 N0842

Date: 2018-11-10

ISO/IEC TR 24772–5

Notes on this document

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 6 – Vulnerability descriptions for the programming language SPARK

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

*Élément introductif — Élément principal — Partie n: Titre de la partie*

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

**Copyright notice**

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

*ISO copyright office*

*Case postale 56, CH-1211 Geneva 20*

*Tel. + 41 22 749 01 11*

*Fax + 41 22 749 09 47*

*E-mail copyright@iso.org*

*Web www.iso.org*

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents

[Foreword viii](#_Toc531005201)

[Introduction ix](#_Toc531005202)

[1. Scope 1](#_Toc531005203)

[2. Normative references 1](#_Toc531005204)

[3. Terms and definitions, symbols and conventions 1](#_Toc531005205)

[3.1 Terms and definitions 1](#_Toc531005206)

[4. Language concepts 2](#_Toc531005207)

[5. Avoiding programming language vulnerabilities in Spark 3](#_Toc531005208)

[6. Specific Guidance for C++ Vulnerabilities 5](#_Toc531005209)

[6.1 General 5](#_Toc531005210)

[6.2 Type System [IHN] 5](#_Toc531005211)

[6.3 Bit Representations [STR] 5](#_Toc531005212)

[6.4 Floating-point Arithmetic [PLF] 6](#_Toc531005213)

[6.5 Enumerator Issues[CCB] 6](#_Toc531005214)

[6.6 Conversion Errors [FLC] 7](#_Toc531005215)

[6.7 String Termination [CJM] 8](#_Toc531005216)

[6.8 Buffer Boundary Violation [HCB] 8](#_Toc531005217)

[6.9 Unchecked Array Indexing [XYZ] 8](#_Toc531005218)

[6.10 Unchecked Array Copying [XYW] 8](#_Toc531005219)

[6.11 Pointer Type Conversions [HFC] 9](#_Toc531005220)

[6.12 Pointer Arithmetic [RVG] 9](#_Toc531005221)

[6.13 NULL Pointer Dereference [XYH] 9](#_Toc531005222)

[6.14 Dangling Reference to Heap [XYK] 9](#_Toc531005223)

[6.15 Arithmetic Wrap-around Error [FIF] 9](#_Toc531005224)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 9](#_Toc531005225)

[6.17 Choice of Clear Names [NAI] 10](#_Toc531005226)

[6.18 Dead Store [WXQ] 10](#_Toc531005227)

[6.19 Unused Variable [YZS] 11](#_Toc531005228)

[6.20 Identifier Name Reuse [YOW] 11](#_Toc531005229)

[6.21 Namespace Issues [BJL] 11](#_Toc531005230)

[6.22 Initialization of Variables [LAV] 12](#_Toc531005231)

[6.23 Operator Precedence and Associativity [JCW] 12](#_Toc531005232)

[6.24 Side-effects and Order of Evaluation of Operands [SAM] 12](#_Toc531005233)

[6.25 Likely Incorrect Expression [KOA] 12](#_Toc531005234)

[6.26 Dead and Deactivated Code [XYQ] 13](#_Toc531005235)

[6.27 Switch Statements and Static Analysis [CLL] 14](#_Toc531005236)

[6.28 Demarcation of Control Flow [EOJ] 15](#_Toc531005237)

[6.29 Loop Control Variables [TEX] 15](#_Toc531005238)

[6.30 Off-by-one Error [XZH] 15](#_Toc531005239)

[6.31 Structured Programming [EWD] 16](#_Toc531005240)

[6.32 Passing Parameters and Return Values [CSJ] 16](#_Toc531005241)

[6.33 Dangling References to Stack Frames [DCM] 16](#_Toc531005242)

[6.34 Subprogram Signature Mismatch [OTR] 16](#_Toc531005243)

[6.35 Recursion [GDL] 17](#_Toc531005244)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 17](#_Toc531005245)

[6.37 Type-breaking Reinterpretation of Data [AMV] 18](#_Toc531005246)

[6.38 Deep vs. Shallow Copying [YAN] 18](#_Toc531005247)

[6.39 Memory Leak and Heap Fragmentation [XYL] 18](#_Toc531005248)

[6.40 Templates and Generics [SYM] 18](#_Toc531005249)

[6.41 Inheritance [RIP] 19](#_Toc531005250)

[6.41.2 Guidance to language users 19](#_Toc531005251)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 19](#_Toc531005252)

[6.43 Redispatching [PPH] 19](#_Toc531005253)

[6.44 Polymorphic variables [BKK] 19](#_Toc531005254)

[6.45 Extra Intrinsics [LRM] 19](#_Toc531005255)

[6.46 Argument Passing to Library Functions [TRJ] 20](#_Toc531005256)

[6.47 Inter-language Calling [DJS] 20](#_Toc531005257)

[6.47 Dynamically-linked Code and Self-modifying Code [NYY] 21](#_Toc531005258)

[6.49 Library Signature [NSQ] 21](#_Toc531005259)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 21](#_Toc531005260)

[6.51 Pre-processor Directives [NMP] 22](#_Toc531005261)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 22](#_Toc531005262)

[6.53 Provision of Inherently Unsafe Operations [SKL] 22](#_Toc531005263)

[6.54 Obscure Language Features [BRS] 23](#_Toc531005264)

[6.55 Unspecified Behaviour [BQF] 23](#_Toc531005265)

[6.56 Undefined Behaviour [EWF] 24](#_Toc531005266)

[6.57 Implementation–defined Behaviour [FAB] 24](#_Toc531005267)

[6.58 Deprecated Language Features [MEM] 26](#_Toc531005268)

[6.59 Concurrency – Activation [CGA] 26](#_Toc531005269)

[6.60 Concurrency – Directed termination [CGT] 26](#_Toc531005270)

[6.61 Concurrent Data Access [CGX] 26](#_Toc531005271)

[6.62 Concurrency – Premature Termination [CGS] 27](#_Toc531005272)

[6.63 Protocol Lock Errors [CGM] 27](#_Toc531005273)

[6.64 Uncontrolled Format String [SHL] 27](#_Toc531005274)

[7. Language specific vulnerabilities for C 27](#_Toc531005275)

[8. Implications for standardization 27](#_Toc531005276)

[Bibliography 28](#_Toc531005277)

[Index 31](#_Toc531005278)

# Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard (“state of the art”, for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-X, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

# Introduction

This Technical Report provides guidance for the programming language C++, so that application developers considering C++ or using C++ will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the C++ language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This report can also be used in comparison with companion Technical Reports and with the language-independent report, TR 24772–1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772–1, which discusses programming language vulnerabilities in a language independent fashion. It is also intended to be used with TR 24772-3, which discusses how the vulnerabilities introduced in TR 24772-1 are manifested in C, which is a subset of C++.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

**Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Vulnerability descriptions for the programming language SPARK**

# 1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this Technical Report document the way that the vulnerability described in the language-independent TR 24772–1 are manifested in C++.

# 2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

TBD

# 3. Terms and definitions, symbols and conventions

## 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, in TR 24772–1, in 14882:2014 and the following apply. Other terms are defined where they appear in *italic* type.

See *C.1 Identification of standards and associated documentation*, plus the references below. In the body of this annex, the following documents are referenced using the short abbreviation that introduces each document, optionally followed by a specific section number. For example “[SLRM 5.2]” refers to section 5.2 of the SPARK Language Definition.

[SLRM] [SPARK Language Definition](http://www.altran-praxis.com/sparkTechnicalReferences.aspx): “SPARK95: The SPADE Ada Kernel (Including RavenSPARK)” Latest version always available from www.altran-praxis.com.

[SB] “High Integrity Software: The SPARK Approach to Safety and Security.” John Barnes. Addison-Wesley, 2003. ISBN 0-321-13616-0.

[IFA] “Information-Flow and Data-Flow Analysis of while-Programs.” Bernard Carré and Jean-Francois Bergeretti, ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 7 No. 1, January 1985. pp 37-61.

[LSP] “A behavioral notion of subtyping.” Barbara Liskov and Jeannette Wing. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6 (November 1994), pp. 1811 - 1841.

# 4. Language concepts

The SPARK language is a contractualized subset of Ada, specifically designed for high-assurance systems. SPARK is designed to be amenable to various forms of static analysis that prevent or mitigate the vulnerabilities described in this TR.

Many terms and concepts applicable to Ada also apply to SPARK. See *C.2 General terminology and concepts*.

This section introduces concepts and terminology which are specific to SPARK and/or relate to the use of static analysis tools.

**Soundness**

This concept relates to the absence of false-negative results from a static analysis tool. A false negative is when a tool is posed the question “Does this program exhibit vulnerability X?” but incorrectly responds “no.” Such a tool is said to be**unsound** for vulnerability X. A sound tool effectively finds **all** the vulnerabilities of a particular class, whereas an unsound tool only finds some of them.

The provision of soundness in static analysis is problematic, mainly owing to the presence of unspecified and undefined features in programming languages. Claims of soundness made by tool vendors should be carefully evaluated to verify that they are reasonable for a particular language, compilers and target machines. Soundness claims are always underpinned by assumptions (for example, regarding the reliability of memory, the correctness of compiled code and so on) that should also be validated by users for their appropriateness.

Static analysis techniques can also be **sound in theory** – where the mathematical model for the language semantics and analysis techniques have been formally stated, proved, and reviewed – but **unsound in practice** owing to defects in the implementation of analysis tools. Again, users should seek evidence to support any soundness claim made by language designers and tool vendors. A language which is **unsound in theory** can never be sound in practice.

The single overriding design goal of SPARK is the provision of a static analysis framework which is **sound in theory,** and as **sound in practice** as is reasonably possible.

In the subsections below, we say that SPARK **prevents** a vulnerability if supported by a form of static analysis which is sound in theory. Otherwise, we say that SPARK **mitigates** a particular vulnerability.

**SPARK Processor**

We define a “SPARK Processor” to be a tool that implements the various forms of static analysis required by the SPARK language definition. Without a SPARK Processor, a program cannot reasonably be claimed to be SPARK at all, much in the same way as a compiler checks the static semantic rules of a standard programming language.

In SPARK, certain forms of analysis are said to be **mandatory** – they are required to be implemented and programs must pass these checks to be valid SPARK. Examples of mandatory analyses are the enforcement of the SPARK language subset, static semantic analysis (e.g. enhanced type checking) and information flow analysis [IFA].

Some analyses are said to be **optional** – a user may choose to enable these additional analyses at their discretion. The most notable example of an optional analysis in SPARK is the generation of verification conditions that will be processed by the analysis and proof tools. Sometimes default SPARK proofs will be assisted by adding static information such as type invariance clauses, assertions, loop invariances and subprogram preconditions and postconditions.

Optional analyses may provide greater depth of analysis, protection from additional vulnerabilities, and functional proofs of correctness.

**Failure modes for static analysis**

Unlike a language compiler, a user can always choose not to run a static analysis tool. Therefore, there are two modes of failure that apply to all vulnerabilities:

1. The user fails to apply the appropriate static analysis tool to their code.
2. The user fails to review or mis-interprets the output of static analysis.

During the static analysis phase, the SPARK language processor generates verification conditions that must be discharged using the SPARK prover. Some proofs require annotations to be added into the program source code to assist the proofs.

Unsafe Programming: In recognition of the occasional need to step outside the type system or to perform “risky” operations, SPARK provides clearly identified language features to do so. Examples include

* Using the generic Unchecked\_Conversion for unsafe type-conversions, and
* Hiding a unit from the spark verification system, by NOT providing the aspect “with SPARK\_MODE” on a unit or on its body.

The **pragma** Suppress allows an implementation to omit certain run-time checks, although the SPARK processor will continue to generate verification conditions to show the correctness of the operation.

# 5. Avoiding programming language vulnerabilities in Spark

In addition to the generic programming rules from TR 24772-1 clause 5.4, additional rules from this section apply specifically to the Spark programming language. The recommendations of this section are restatements of recommendations from clause 6, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

Every guidance provided in this section, and in the corresponding Part section, is supported by material in Clause 6 of this document, as well as other important recommendations.

***TBD***

|  |  |  |
| --- | --- | --- |
| Index |  | Reference |
| 1 |  |  |
| 2 |   |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |

#

Need to consider C++-11, 14 and 17.

# 6. Specific Guidance for SPARK Vulnerabilities

## 6.1 General

This clause contains specific advice for SPARK about the possible presence of vulnerabilities as described in TR 24772-1, and provides specific guidance on how to avoid them in SPARK code. This section mirrors TR 24772-1 clause 6 in that the vulnerability “Type System [IHN]” is found in 6.2 of TR 24772–1, and C++ specific guidance is found in clause 6.2 and subclauses in this TR.

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

SPARK’s type system is a simplification of Ada’s type system. Both explicit and implicit conversions are permitted in SPARK, as is instantiation and use of Unchecked\_Conversion [SB 1.3]. Developers can choose to use the underlying types such as full integers, floating point numbers, characters and strings instead of much more tightly specified data types and can use the less safe conversions. Even when using these less safe constructs, users can use the Spark language precondition, postcondition, invariance mechanisms and the static provers to eliminate almost all of the vulnerabilities discussed in TR 24772-1 clause 6.2.

SPARK mitigates the vulnernabilities discussed in TR 24772-1 clause 6.2 through the use of its very strong typing system, as well as a strong contract model useful for developing formal proofs of correctness, and a strong proof tool to verification the type safety of the complete program.

 A design goal of SPARK is the provision of *static type safety,* meaning that programs can be shown to be free from all run-time type failures using entirely static analysis. If this optional analysis is achieved, a SPARK program should never raise an exception at run-time.

The SPARK compiler generates verification conditions that are discharged by the verification tools. Failure to execute the verification tools does not prevent the compiler and linker from generating executables from legal programs, so developers are responsible for ensuring that executables are only produced for code that has also successfully completed data flow analysis and verification.

### 6.2.2 Guidance to language users

* Follow the guidance of Tr 24772-1 clause 6.2.2.
* Use the SPARK analysis and proof tools to verify the absence of runtime errors.

## 6.3 Bit Representations [STR]

### 6.3.1 Applicability to language

SPARK mitigates this vulnerability.

SPARK provides a semantics which is independent of the underlying representation chosen by a compiler for a particular target machine. Representation clauses are permitted, but these do not affect the semantics as seen by a static analysis tool [SB 1.3].

### 6.3.2 Guidance to language users

* Explicitly document any reliance on bit ordering or usage using SPARK’s representation clauses.
* Where bit ordering can change either between the development host and the target, or between interfaced targets, provide compatible types with derived types that document each system’s mapping and explicitly convert between them.
* Localize and document the code associated with explicit manipulation of bits and bit fields.
* Use Spark’s static analysis tools and proof tools to verify the correct usage and conversion between types.

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

SPARK specifies adherence to the IEEE Floating Point Standards (ISO/IEC/IEEE-60559-2011, IEEE-854-1987).

The vulnerability in SPARK is as described in subclause 6.4.2 of TR 24772-1.

### 6.4.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.4.5 of TR 24772-1.
* Rather than using predefined types, such as Float and Long\_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, **digits** 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
* Avoid comparing floating-point values for equality. Instead, use comparisons that account for the approximate results of computations. Consult a numeric analyst when appropriate.
* Make use of static arithmetic expressions and static constant declarations when possible, since static expressions in SPARK are computed at compile time with exact precision.
* Use mathematical models and SPARK’s proof tools to verify the correctness of mathematical calculations in floating point. This may necessitate recasting algorithms to make them amenable to such proofs.
* Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of the predefined floating-point attributes (such as 'Exponent).

## 6.5 Enumerator Issues[CCB]

### 6.5.1 Applicability to language

Enumeration representation specification may be used to specify non-default representations of an enumeration type, for example when interfacing with external systems. All of the values in the enumeration type must be defined in the enumeration representation specification. The numeric values of the representation must preserve the original order. For example:

**type** IO\_Types **is** (Null\_Op, Open, Close, Read, Write, Sync);

**for** IO\_Types **use** (Null\_Op => 0, Open => 1, Close => 2,

Read => 4, Write => 8, Sync => 16);

An array may be indexed by such a type. SPARK does not prescribe the implementation model for arrays indexed by an enumeration type with non-contiguous values. Two options exist: Either the array is represented “with holes” and indexed by the values of the enumeration type, or the array is represented contiguously and indexed by the position of the enumeration value rather than the value itself. In the former case, the vulnerability described in TR 24772-1 subclause 6.5 exists only if unsafe programming is applied to access the array or its components outside the protection of the type system. Within the type system, the semantics are well defined and safe. The vulnerability of unexpected but well-defined program behaviour upon extending an enumeration type exists in SPARK. In particular, subranges or **others** choices in aggregates and case statements are susceptible to unintentionally capturing newly added enumeration values.

### 6.5.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.5.5 of TR 24772-1. In particular, use SPARK’s analysis and proof tools to diagnose inappropriate use of enumeration types or values.
* For **case** statements and aggregates, do not use the **others** choice.
* For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition

## 6.6 Conversion Errors [FLC]

### 6.6.1 Applicability to language

SPARK is designed to be amenable to static verification of the absence of predefined exceptions, and in particular all cases covered by this vulnerability [SB 11]. All numeric conversions (both explicit and implicit) give rise to verification conditions that are discharged using SPARK’s automated theorem-prover. Except for the unsafe generic function Unchecked\_Conversion, conversion between non-numeric types can only happen

* if one type is a derivation of the other,
* if both types are subtypes of a common parent, or
* if all components of the source and target types are either numeric types or related types and conversion is done component-by-component.

In these cases, SPARK will generate the respective verification conditons to be discharged by the toolchain or by the user.

If Unchecked\_Conversion is used, SPARK will assume that the conversion is correct and will generate TRUE conditions for the conversion and for ‘Valid applied to the conversion. Unchecked conversions are highly dependent on the layout of the source and targets of the conversion as well as values contained and do not fit the Spark models analysis. Therefore, static correctness of unchecked conversions must be verified by other means.

### 6.6.2 Guidance to language users

* Use Spark’s analysis and proof tools to staticaly verify the absense of errors in the use of conversions.
* Create contract models and Sparks proof tools to verify the correct functional use of conversions.
* If Unchecked\_Conversion is used, use other analysis methods to verify the correctness of the conversion(s).

## 6.7 String Termination [CJM]

This vulnerability is not applicable to SPARK as strings are not delimited by a termination character. Spark programs that interface to languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

## 6.8 Buffer Boundary Violation [HCB]

With the exception of unsafe programming (see 4 Language concepts), this vulnerability is not applicable to Spark (see 6.9 Unchecked Array Indexing [XYZ] and 6.10 Unchecked Array Copying [XYW]).

## 6.9 Unchecked Array Indexing [XYZ]

### 6.9.1 Applicability to language

SPARK static analysis verifies the absence of boundary violations discussed in TR 24772-1 clause 6.9, through techniques such as theorem proving or abstract interpretation [SB 11].

SPARK programs that have been subject to this level of analysis can be compiled with run-time checks suppressed, supported by a body of evidence that such checks could never fail, and thus removing the possibility of erroneous execution.

### 6.9.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.9.5 of TR 24772-1.
* Use SPARK’s support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.
* Use SPARK’s verification tools and use contracts to verify the functional correctness of the code.

## 6.10 Unchecked Array Copying [XYW]

SPARK prevents this vulnerability.

Array assignments in SPARK are only permitted between objects that have statically matching bounds. Hence all violations are detected at compile time.

## 6.11 Pointer Type Conversions [HFC]

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access types [SB 1.3, SLRM 3.10].

## 6.12 Pointer Arithmetic [RVG]

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access types [SB 1.3, SLRM 3.10].

## 6.13 NULL Pointer Dereference [XYH]

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access types [SB 1.3, SLRM 3.10].

## 6.14 Dangling Reference to Heap [XYK]

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access types [SB 1.3, SLRM 3.10].

## 6.15 Arithmetic Wrap-around Error [FIF]

### 6.15.1 Applicability to language

With the exception of unsafe programming (see [4 Language concepts](#_4_Language_concepts)), this vulnerability is not applicable to SPARK as wrap-around arithmetic is limited to modular types. Arithmetic operations on such types use modulo arithmetic, and thus no such operation can create an invalid value of the type.

For non-modular arithmetic, the predefined exception Constraint\_Error is raised whenever a wrap-around occurs but implementations are allowed to refrain from doing so when a correct final value is obtained.

### 6.15.2 Guidance to language users

* Use the SPARK static analysis tools to show that exceptions cannot be raised by values exceeding their specified limits.
* Develop contracts and use SPARK analysis and prover to verify that the program meets the specified contracts.

## 6.16 Using Shift Operations for Multiplication and Division [PIK]

With the exception of unsafe programming (see [4 Language concepts](#_4_Language_concepts)), this vulnerability is not applicable to Spark as shift operations are limited to the modular types declared in the standard package Interfaces, which are not signed entities.

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

There are two possible issues: the use of the identical name for different purposes (overloading) and the use of similar names for different purposes.

This vulnerability does not address overloading, which is covered in 6.20 Identifier Name Reuse [YOW].

The risk of confusion by the use of similar names might occur through:

* Mixed casing. This is not an issue since SPARK treats upper and lower case letters in names as identical. Confusion for the progrmmer may arise through an attempt to use Item and ITEM as distinct identifiers with different meanings, but the language system and strong type checking will ensure appropriate and correct usage.
* Underscores and periods. SPARK permits single underscores in identifiers and they are significant. Thus BigDog and Big\_Dog are different identifiers. But multiple underscores (which might be confused with a single underscore) leading underscores and trailing underscores are forbidden. Periods in SPARK denote substructures and hence are meaningful.
* Singular/plural forms. SPARK permits the use of identifiers which differ solely in this manner such as Item and Items. The programmer may create plural and singular forms to identify single items or collections, but the language system and strong type checking will ensure appropriate and correct usage.
* International character sets. SPARK strictly conform to the appropriate International Standard for character sets.
* Identifier length. All characters in an identifier in SPARK are significant. And an identifier cannot be split over the end of a line. The only restriction on the length of an identifier is that enforced by the line length and this is guaranteed by the language standard to be no less than 200.

SPARK permits the use of names such as X, XX, and XXX (which might all be declared as integers) and a programmer could easily, by mistake, write XX where X (or XXX) was intended. SPARK does not attempt to catch such errors unless the developer creates contracts that define the functional behaviour of the code module and uses the analysis and proof tools to verify correct usage.

### 6.17.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.17.5 of TR 24772-1.
* Develop contracts and use the SPARK analysis and proof tools to verify that the program meets the specified contracts.
* Avoid the use of similar names to denote different objects of the same type.
* Adopt a project convention for dealing with similar names

## 6.18 Dead Store [WXQ]

SPARK prevents this vulnerability through automatic static information flow analysis, which detects dead stores. Additionally, SPARK requires variables that are used for output to the environment, where multiple writes to a variable without intervening reads could be confused as dead store, to be specifically identified. In this case, the information flow analysis for such variables is modified since it is known that consecutive writes to such variables might not constitute a dead store.

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

SPARK is designed to permit sound static analysis of the following cases for information flow analysis:

* Variables which are declared but not used at all.
* Variables which are assigned to, but the resulting value is not used in any way that affects an output of the enclosing subprogram. This is called an “ineffective assignment” in SPARK

### 6.19.2 Guidance to language users

* Mark variables that are written by a subprogram but read elsewhere with the aspect Volatile or Volatile\_Components.
* Follow the guidance of SPARK flow analysis with respect to unused variables.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

SPARK permits local scope, and names within nested scopes, including declarative items in **for** loops. Local names can hide identical names declared in an outer scope. As such it is susceptible to the vulnerability described in TR 24772-1 clause 6.20 [YOW]. For subprograms and other overloaded entities the problem is reduced by the fact that hiding also takes the signatures of the entities into account. Entities with different signatures, therefore, do not hide each other.

Name collisions with keywords cannot happen in SPARK because keywords are reserved.

The mechanism of failure identified in subclause 6.20.3 of TR 24772-1 regarding the declaration of non-unique identifiers in the same scope cannot occur in Spark because all characters in an identifier are significant.

### 6.20.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.20.5 of TR 24772-1.
* Use *expanded names* whenever confusion may arise*.*
* Use Spark compilers and static analysis tools to detect declarations in inner scopes that hide declarations in outer scopes

## 6.21 Namespace Issues [BJL]

This vulnerability is not applicable to SPARK, since the language does not attempt to disambiguate conflicting names imported from different packages. Use of a name with conflicting imported declarations causes a compile time error. The programmer can disambiguate the name usage by using a expanded name that identifies the exporting package.

## 6.22 Initialization of Variables [LAV]

SPARK prevents this vulnerability through mandatory static information flow analysis.

### *Question – does spark flow analysis failures halt the compilation process? NO.*

## 6.23 Operator Precedence and Associativity [JCW]

### 6.23.1 Applicability to language

Since this vulnerability is about "incorrect beliefs" of programmers, there is no way to establish a limit to how far incorrect beliefs can go. However, SPARK is less susceptible to that vulnerability than many other languages, since

* There are six levels of precedence, and associativity is close to common expectations. For example, an expression like A = B or C = D will be parsed as expected, as (A = B) or (C = D).
* Mixed logical operators are not allowed without parentheses, for example, "A or B or C" is valid, as well as "A and B and C", but "A and B or C" is not; the user must write "(A and B) or C" or "A and (B or C)".
* Assignment is not an operator.

### 6.23.2 Guidance to language users

* Follow the guidance provided in TR 24772-1 clause 6.23.5
* Use parentheses any time arithmetic operators, logical operators, mixed logical operators such as and and and then and shift operators are mixed in an expression.
* Create contracts that specify the expressions in mathematical terms and verify using the Spark static analysis tools.

## 6.24 Side-effects and Order of Evaluation of Operands [SAM]

This vulnerability is prevented by SPARK since it provides a number of mitigations to prevent erroneous behaviour from side effects or order of evaluation:

* There are no operators that have direct side effects on their operands using the language-defined operations, especially not the increment and decrement operation.
* SPARK does not permit multiple assignments in a single expression or statement.
* SPARK functions are side-effect free.

## 6.25 Likely Incorrect Expression [KOA]

### 6.25.1 Applicability to language

An instance of this vulnerability consists of two syntactically similar constructs such that the inadvertent substitution of one for the other may result in a program which is accepted by the compiler but does not reflect the intent of the author.

The examples given in subclause 6.25 of TR 24772-1 are not problems in SPARK because of the strong typing and because an assignment is not an expression in SPARK.

In SPARK, a type-conversion and a qualified expression are syntactically similar, differing only in the presence or absence of a single character:

 Type\_Name (Expression) -- a type-conversion

vs.

 Type\_Name'(Expression) -- a qualified expression

Typically, the inadvertent substitution of one for the other results in either a semantically incorrect program which is rejected by the compiler or in a program which behaves in the same way as if the intended construct had been written. In the case of a constrained array subtype, the two constructs differ in their treatment of sliding (conversion of an array value with bounds 100 .. 103 to a subtype with bounds 200 .. 203 will succeed; qualification will fail a run-time check).

Potential task-based difficulties in Ada are avoided in SPARK because SPARK only supports the Ravenscar Tasking Profile wich removes order of access ambiguities

Problems arising from a faiure to use short-circuit Boolean forms almost never arise in Spark programs because access types, which are the largest driver of the need for short-circiut Boolean forms, are forbidden.

### 6.25.2 Guidance to language users

* If a possible need for short-circuit Booleans is identified, construct contracts that fully express the logic required, for example

if I < N and completed(X[I}) then . . . – should have been “and then”

 assert (completed(X[i] => I<N)

end if;

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

SPARK provides static analysis of dead paths. A dead path cannot be executed in that the combination of conditions for its execution are logically equivalent to *false.* Such cases can be statically detected by theorem proving in SPARK.

### 6.26.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.26.5 of TR 24772-1.
* Use the Spark prover to identify non-executable paths.
* Use implementation-specific mechanisms, if provided, to support the elimination of dead code. In some cases, use pragmas such as Restrictions, Suppress, or Discard\_Names to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example:

**package** Pkg **is**

**type** Enum **is** (Aaa, Bbb, Ccc);

**pragma** Discard\_Names( Enum );

**end** Pkg;

If Pkg.Enum'Image and related attributes (e.g., Value, Wide\_Image) of the type Enum are never used, and if the implementation normally builds a table of the enumeration literals, then the **pragma** allows the elimination of the table.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

With the exception of unsafe programming (see [4 Language concepts](#_4_Language_concepts)) and the use of default cases, this vulnerability is not applicable to SPARK, which ensures that a case statement provides exactly one alternative for each value of the expression's subtype. This restriction is enforced at compile time.

The **others** clause may be used as the last choice of a case statement to capture any remaining values of the case expression type that are not covered by the preceding case choices. If the value of the expression is outside of the range of this subtype (e.g., due to an uninitialized variable), then the resulting behaviour is well-defined (Constraint\_Error is raised). Control does not flow from one alternative to the next. Upon reaching the end of an alternative, control is transferred to the end of the **case** statement.

The remaining vulnerability is that unexpected values can be captured by the **others** clause or a subrange as case choice. For example, when the range of the type Character was extended from 128 characters to the 256 characters in the Latin-1 character type, an **others** clause for a **case** statement with a Character type case expression originally written to capture cases associated with the 128 characters type now also captures the 128 additional cases introduced by the extension of the type Character. Some of the new characters may have needed to be covered by the existing case choices or new case choices.

Another example is the inclusion of additional values internal to a range (usually done by adding an enumeration value to an enumeration type but not at the first or last of that type), and some case statements choices hide the addition in a range of choices.

### 6.27.2 Guidance to language users

* For **case** statements and aggregates, avoid the use of the **others** choice.
* For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.15F[[1]](#footnote-1)
* When adding enumeration values to an enumeration type, review all of the places where if statements or case choices are used to ensure that the position of the added value does not create logic errors.

## 6.28 Demarcation of Control Flow [EOJ]

This vulnerability does not apply to SPARK, since SPARK enforces a clear demarcation of all branching control flows, if statements, case statements, loops, and blocks.

## 6.29 Loop Control Variables [TEX]

With the exception of unsafe programming (see [4 Language concepts](#_4_Language_concepts)), this vulnerability is not applicable to SPARK, which defines a **for loop** where the number of iterations is controlled by a loop control variable (called a loop parameter). This value has a constant view and cannot be updated within the sequence of statements of the body of the loop.

## 6.30 Off-by-one Error [XZH]

Therefore part of the documented in TR 24772-1 clause 6.30

### 6.30.1 Applicability to language

#### Confusion between the need for < and <= or > and >= in a test.

A SPARK **for loop** does not require the programmer to specify a conditional test for loop termination. Instead, the starting and ending value of the loop are specified which eliminates this source of off-by-one errors. There are also special **for loop** structures that iterate through an entire array or container. These avoid the need to specify any bounds for the iteration. A **while loop** however, lets the programmer specify the loop termination expression, which could be susceptible to an off-by-one error. Any off-by-one error that gives rise to the potential for a buffer-overflow, range violation, or any other construct that could give rise to a predefined exception, will be detected by static analysis in SPARK

#### Confusion as to the index range of an algorithm.

Although there are language defined attributes to symbolically reference the start and end values for a loop iteration, the language does allow the use of explicit values and loop termination tests. Off-by-one errors can result in these circumstances.

Care should be taken when using the 'Length attribute in the loop termination expression. The expression should generally be relative to the 'First value.

SPARK’s strong typing eliminates the potential for buffer overflow associated with this vulnerability. In addition, SPARK’s static analysis will detect erroneous uses of loops that do not properly cover a range.

If the error is not statically caught at compile time, then a run-time check generates an exception if an attempt is made to access an element outside the bounds of an array.

### 6.30.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.30.5 of TR 24772-1.
* Whenever possible, use a **for loop** instead of a **while loop**.
* Whenever possible, use the form of iteration that takes the name of the array or container and nothing more.
* When indices are necessary, use the 'first, 'last, and 'range attributes for loop termination, e.g. **for** I **in** My\_Array'range **loop**….
* If the 'length attribute must be used, ensure that the index computation considers the starting index value for the array.
* Use the SPARK analysis and proof tools on all code and do not build anexecuting program until all proofs pass.

## 6.31 Structured Programming [EWD]

### 6.31.1 Applicability to language

SPARK programs can exhibit many of the vulnerabilities noted in Subclause 6.31 of TR 24772-1: leaving a **loop** at an arbitrary point, and multiple exit points from subprograms. SPARK does not provide the ability to perform non-local jumps or to have multiple entries to subprograms.

SPARK provide mitigations for these issues through the use of loop invariance and loop termination contracts.

### 6.31.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.31.5 of TR 24772-1.
* Create SPARK contracts to verify that code written comforms to the its functional specification.

**PROVIDE EXAMPLES OF LOOP PROOFS**

 **- see** <http://docs.adacore.com/spark2014-docs/html/ug/en/tutorial.html>

 <http://docs.adacore.com/spark2014-docs/html/ug/gnatprove_by_example/loop.html>

## 6.32 Passing Parameters and Return Values [CSJ]

This vulnerability is not applicable to SPARK since SPARK functions cannot have side effects, and since procedure and entry parameters must always be declared as **in**, **out**, or **in out** and access types are forbidden, eliminating the possible use of indirection in parameters.

## 6.33 Dangling References to Stack Frames [DCM]

Access types (pointers) are forbidden in Spark, hence this vulnerability does not apply to Spark.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

Except for the case of calls to/from subprograms where the other side is a foreign language, or the case where a Spark generic subprogram or subprogram of a generic package contains formal parameters with default expressions, this vulnerability does not apply.

The first case, for interlanguage calls is addressed in 6.46.

In the second case, actual parameters are constructed for the missing formal parameters via the default expression, hence all subprogram expressions will exist and there will be no stack corruption will occur.

At compilation time, the parameter association is checked to ensure that the type of each actual parameter matches the type of the corresponding formal parameter. In addition, the formal parameter specification may include default expressions for a parameter. Hence, a procedure call may be constructed with some actual parameters missing. In this case, if there is a default expression for the missing parameter, then the call will be compiled without any errors. If no default expression exists for missing parameters, then an compilation error is generated.

### 6.34.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.34.5 of TR 24772-1.
* Minimize the use of default expressions for formal parameters.

*Any additional guidance here?*

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

SPARK permits recursion. The exception Storage\_Error is raised when the recurring execution results in insufficient storage. This will result in program termination unless an exception handler is placed outside the SPARK portion of the program.

### 6.35.2 Guidance to language users

* Apply the guidance described in TR 24772-1 clause 6.35.5.
* *Use contracts and assertions to guarantee that each recursive call is a reduction from the previous call, and to verify that all recursive calls are bounded.*
* Use the asynchronous control construct to time the execution of a recurring call and to terminate the call if the time limit is exceeded.
* Consider applying the restriction No\_Recursion or No\_Reentrancy to eliminate this vulnerability.

## 6.36 Ignored Error Status and Unhandled Exceptions [OYB]

### 6.36.1 Applicability to language

SPARK permits the declaration of exceptions, and the execution of the **raise** statement. SPARK does not permit exception handlers, which means that all SPARK programs must be verified to be free of all predefined and user defined exceptions.

The ‘Valid attribute can be used to verify the result of Unchecked\_Conversion and to handle error conditions, since the verification tools will assume that the result is always valid.

### 6.36.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.36.5 of TR 24772-1.
* Use the SPARK flow static analysis to verify the absence of runtime errors.
* Create and statically verify contracts to verify that error situations that can lead to exceptions do not ccur.
* Use the result of the 'Valid attribute to check for the validity of values delivered to an SPARK program from an external device prior to use or from Unchecked\_Conversion and explicitly handle both TRUE and FALSE cases..
* Consider placing a top-level exception handler in the main program (external to SPARK) and in each task so that notification of failure can be given.

## 6.37 Type-breaking Reinterpretation of Data [AMV]

### 6.37.1 Applicability to language

SPARK permits the instantiation and use of Unchecked\_Conversion as in Ada. The result of a call to Unchecked\_Conversion cannot be assumed to be valid, so static verification tools must be used to validate of the result before further analysis can succeed or the ‘valid cosntruct can be used at runtime inside an if statement with verified paths to handle the case of valid conversion or of invalid conversion.

Language rules prevent the changing of a discriminate of a variable unless the whole object is written, so reinterpreting an objects components is not possible. Record extensions require that the extension components be written or read by subprograms with visibility to the extensions, hence those elements will be correctly interpreted.

### 6.37.2 Guidance to language users

* Follow the guidelines of TR 24772-1 clause 6.38.5.
* Consider applying the restrictions No\_Use\_Of\_Pragma(Unchecked\_Union),
No\_Use\_Of\_Aspect(Unchecked\_Union), and No\_Unchecked\_Conversion to ensure this vulnerability cannot arise.

## 6.38 Deep vs. Shallow Copying [YAN]

This vulnerability does not apply to SPARK since it does not permit the use of access types.

## 6.39 Memory Leak and Heap Fragmentation [XYL]

This vulnerability does not apply to Spark since Spark does not permit the use of access types.

## 6.40 Templates and Generics [SYM]

With the exception of unsafe programming (see [4 Language concepts](#_4_Language_concepts)), this vulnerability is not applicable to SPARK since its generics model is based on imposing a contract on the structure and operations of the types that can be used for instantiation. Also, explicit instantiation of the generic is required for each particular type and SPARK generates static checks for each instantiation of the generic.

Therefore, the compiler is able to check the generic body for programming errors, independently of actual instantiations. At each actual instantiation, the compiler will also check that the instantiated type meets all the requirements of the generic contract.

SPARK also does not allow for ‘special case’ generics for a particular type, therefore behaviour is consistent for all instantiations.

## 6.41 Inheritance [RIP]

### 6.41.1 Applicability to language

The vulnerability documented in TR 24772-1 subclause 6.41 applies to Spark.

SPARK permits a restricted form of multiple inheritance, where only one of the multiple ancestors (the parent) may implement operations. All other ancestors (interfaces) can only specify the operations’ signature, and whether the operation must be overridden, or can simply do nothing if never explicitly defined. Therefore, SPARK does not suffer from multiple inheritance related vulnerabilities.

SPARK has no preference rules to resolve ambiguities of calls on primitive operations of tagged types. Hence the related vulnerability documented in TR 24772-1 subclause 6.41 does not apply.

## 6.41.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.41.5 of TR 24772-1.
* Use the overriding indicators on potentially inherited subprograms to ensure that the intended set of operations are overridden, thus preventing the accidental redefinition or failure to redefine an operation of the parent.
* Specify Pre’Class and Post’Class aspects when a primitive operation is initially defined, to indicate the properties of inputs that any overridings must accept, and the properties of outputs that any overridings must produce.

## 6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP]

This vulnerability does not apply to Spark, since SPARK generates static checks that the Liskov Substitution Principle is followed across the hierarchy, and discharged using the SPARK prover.

## 6.43 Redispatching [PPH]

This vulnerability does not apply to Spark, since SPARK generates static checks that redispatching is correct and appropriate, and the static checks are discharged using the SPARK prover.

## 6.44 Polymorphic variables [BKK]

Except for unsafe programming (see [4 Language concepts](#_4_Language_concepts)), the vulnerabilities related to unsafe casts do not apply to Spark. For downcasts, SPARK generates static checks that redispatching is correct and appropriate, and the static checks are discharged using the SPARK prover.

## 6.45 Extra Intrinsics [LRM]

The vulnerability does not apply to SPARK, because all subprograms, whether intrinsic or not, belong to the same name space. This means that all subprograms must be explicitly declared, and the same name resolution rules apply to all of them, whether they are predefined or user-defined. If two or more subprograms with the same name and signature are visible (that is to say nameable) at the same place in a program, then a call using that name will be rejected as ambiguous by the compiler, and the programmer will have to specify (for example, by means of an expanded name) which subprogram is meant.

## 6.46 Argument Passing to Library Functions [TRJ]

### Applicability to language

The general vulnerability that parameters might have values precluded by preconditions of the called routine applies to SPARK.

To the extent that the preclusion of values can be expressed as part of the type system of SPARK, the preconditions are checked by the compiler statically or dynamically and thus are no longer vulnerabilities. For example, any range constraint on values of a parameter can be expressed in SPARK by means of type or subtype declarations. Type violations are detected at compile time, subtype violations cause run-time exceptions. For that situation, preconditions, postconditions, type invariants, and subtype predicates can be specified explicitly to express more complex restrictions to be observed by callers. These can be recognized by other static analysis tools as part of program verification.

### 6.46.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.46.5 of TR 24772-1.
* Exploit the type and subtype system of SPARK to express restrictions on the values of parameters and results.
* Specify explicit preconditions and postconditions for subprograms wherever practical.
* Specify subtype predicates and type invariants for subtypes and private types when appropriate.
* Execute the SPARK analysis tools and use successful completion as a gate for compleing program build.

## 6.47 Inter-language Calling [DJS]

### 6.47.1 Applicability to language

The vulnerability applies to SPARK, however SPARK provides mechanisms to interface with common languages, such as C, C++, Fortran and COBOL, so that vulnerabilities associated with interfacing with these languages can be avoided.

### 6.47.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.47.5 of TR 24772-1.
* Use the inter-language methods and syntax specified by SPARK and ISO/IEC 8652 [15] when the routines to be called are written in languages that ISO/IEC 8652 [15] specifies an interface with.
* Use interfaces to the C programming language where the other language system(s) are not covered by ISO/IEC 8652, but the other language systems have interfacing to C.
* Make explicit checks on all return values from foreign system code artifacts, for example by using the 'Valid attribute or by performing explicit tests to ensure that values returned by inter-language calls conform to the expected representation and semantics of the SPARK application.

## Dynamically-linked Code and Self-modifying Code [NYY]

With the exception of unsafe programming (see 4 Language concepts), this vulnerability is not applicable to SPARK, which supports neither dynamic linking nor self-modifying code. The latter is possible only by exploiting other vulnerabilities of the language in the most malicious ways and even then it is still very difficult to achieve.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

SPARK provides mechanisms to explicitly interface to modules written in other languages. **Pragma**s Import, Export and Convention permit the name of the external unit and the interfacing convention to be specified.

### Even with the use of pragma Import, pragma Export and pragma Convention the vulnerabilities stated in subclause 6.49 of TR 24772-1 are possible. Names and number of parameters change under maintenance; calling conventions change as compilers are updated or replaced, and languages for which Spark does not specify a calling convention may be used.

### 6.49.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.49.5 of TR 24772-1.

## 6.50 Unanticipated Exceptions from Library Routines [HJW]

### 6.50.1 Applicability to language

SPARK permits the declaration and raising of exceptions, but does not support exception handlers, so any exception raised will cause either the task that was subject to the exception to silently terminate, or the main program to terminate. Since SPARK is a subset of Ada, it is possible to hide the main body of a task or the main subprogram from SPARK and place an exception handler there to perform appropriate notifications or last wishes.

### 6.50.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.50.5 of TR 24772-1.
* Ensure that the interfaces with libraries written in other languages are compatible in the naming and generation of exceptions.
* Consider failure strategies and consider placing exception handlers at the top level of all tasks and the main subprogram.

Note: Since such declarations are external to SPARK, wrapping the main subprogram with another subprogram that exclusively calls the main SPARK subprogram and handles and exception minimizes the amount of non-spark code. Similarly for tasks, placing the task code in a subprogram that never exits and making the task body contain only the call to that subprogram and the exception handlers minimizes the amount of non-spark code.

* Document any exceptions that may be raised by any Ada units being used as library routines.

## 6.51 Pre-processor Directives [NMP]

This vulnerability is not applicable to SPARK, which does not have a pre-processor.

## 6.52 Suppression of Language-defined Run-time Checking [MXB]

### 6.52.1 Applicability to language

The vulnerability exists in SPARK since **pragma** Suppress permits explicit suppression of language-defined checks on a unit-by-unit basis or on partitions or programs as a whole. (The language-defined default, however, is to perform the runtime checks that prevent the runtime vulnerabilities.) **Pragma** Suppress can suppress all language-defined checks or 12 individual categories of checks (see subclause 11.5 of ISO/IEC 8652 [15]). Note, however, that SPARK creates verification conditions to be discharged, even if suppression is used.

### 6.52.2 Guidance to Language Users

Follow the mitigation mechanisms of subclause 6.52.5 of TR 24772-1.

##

## 6.53 Provision of Inherently Unsafe Operations [SKL]

### 6.53.1 Applicability to language

In recognition of the occasional need to step outside the type system or to perform “risky” operations, SPARK provides clearly identified language features to do so. Examples include the generic Unchecked\_Conversion for unsafe type-conversions and mechanisms to implement unit bodies outside of Spark (in Ada).

For Unchecked\_Conversion, the declaring unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units.

For programming in Ada, instead of SPARK, the SPARK processor only provides restrictions and analysis on packages and subprograms (or their bodies) that have the apsect “Spark\_Mode” in the declaration. It is permissible to have the specification in Spark\_Mode and the body not. This provides for callers or users of the unit to have the call checked even if the body is not Spark.

### 6.53.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.53.5 of TR 24772-1. In particular, use the Spark static analysis tools to identify inherently unsafe operations.
* Avoid the use of unsafe programming practices.
* Use the **pragma** Restrictions to prevent the inadvertent use of unsafe language constructs.
* Carefully scrutinize any code that refers to a program unit explicitly designated to provide unchecked operations.
* Use non-SPARK units sparingly, and ensure that a thorough analysis is performed on the code since the SPARK verification tools will not be used.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability of language

SPARK provides facilities for a wide range of application areas. Because some areas are specialized, it is likely that a programmer not versed in a special area might misuse features for that area. For example, the use of tasking features for concurrent programming requires knowledge of this domain.

### 6.54.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.54.5 of TR 24772-1.
* Use the **pragma** Restrictions to prevent the use of obscure features of the language.
* *Similarly, avoid features in a Specialized Needs Annex of ISO/IEC 8652 unless the application area concerned is well-understood. ???*
* Use the language-defined restriction No\_Dependence to prevent the use of specified pre-defined or user-defined libraries.

## 6.55 Unspecified Behaviour [BQF]

### 6.55.1 Applicability of language

In SPARK, there are two main categories of unspecified behaviour, one having to do with unspecified aspects of normal run-time behaviour, and one having to do with *bounded errors*, errors that need not be detected at run-time but for which there is a limited number of possible run-time effects.

For Bounded\_Error, SPARK detects and issues diagnostic messages for all occurrances.

For the normal behaviour category, there is one aspects of run-time behaviour that might be unspecified; the order in which certain actions are performed at run-time. SPARK assumes left-to-right association of operators at the same level of precedence, so is susceptible to implementations that group operations in different orders. Spark language processors can detect such usage when tuned with implementation behaviour information.

### 6.55.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.55.5 of TR 24772-1.

## 6.56 Undefined Behaviour [EWF]

From Ada. Can this be reduced? Some removals (e.g. discussion of access types or address-to-access conversions).

### 6.56.1 Applicability to language

In SPARK, undefined behaviour is called *erroneous execution*, and can arise from certain errors that are not required to be detected by the implementation, and whose effects are not in general predictable.

There are various kinds of errors that can lead to erroneous execution, including:

* Changing a discriminant of a record (by assigning to the record as a whole) while there remain active references to subcomponents of the record that depend on the discriminant;
* Referring via a task id, or tag to an object, task, or type that no longer exists at the time of the reference;
* Sharing an object between multiple tasks without adequate synchronization;
* Suppressing a language-defined check that is in fact violated at run-time;
* Specifying the alignment of an object in an inappropriate way;
* Using Unchecked\_Conversion, or calling an imported subprogram to create a value that has an *abnormal* representation.

Any occurrence of erroneous execution represents a failure situation, as the results are unpredictable, and may involve overwriting of memory, jumping to unintended locations within memory, and other uncontrolled events.

SPARK mitigates most of these cases, however, implementations that need to implement features that can lead to undefined behaviour often step outside of Spark (by leaving “with SPARK\_Mode” off the unit declaration or the unit body declaration) and therefore correct behaviour must be shown by other means.

### 6.56.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.56.5 of TR 24772-1.
* Ensure that all data shared between tasks are either private within a protected object or marked Atomic;
* Use **pragma** Suppress only after the code has been completely analyzed that the SPARK analysis tools with no errors reported. The other errors that can lead to erroneous execution are less common, but clearly in any given Ada application, care must be taken when using features such as:
* Unchecked\_Conversion;
* The results of imported subprograms;
* Discriminant-changing assignments to global variables.
* Minimize the use of removing “with SPARK\_Mode” from unit or unit body declarations and devise alternate verification mechanisms for units that are not analysed by the SPARK processor.

## 6.57 Implementation–defined Behaviour [FAB]

### 6.57.1 Applicability to language

There are a number of situations in Ada where the language semantics are implementation defined, to allow the implementation to choose an efficient mechanism, or to match the capabilities of the target environment. Each of these situations is identified in Annex M of ISO/IEC 8652, and implementations are required to provide documentation associated with each item in Annex M to provide the programmer with guidance on the implementation choices.

A failure can occur in an Ada application due to implementation-defined behaviour if the programmer presumed the implementation made one choice, when in fact it made a different choice that affected the results of the execution. In many cases, a compile-time message or a run-time exception will indicate the presence of such a problem. For example, the range of integers supported by a given compiler is implementation defined. However, if the programmer specifies a range for an integer type that exceeds that supported by the implementation, then a compile-time error will be indicated, and if at run time a computation exceeds the base range of an integer type, then a Constraint\_Error is raised.

Failure due to implementation-defined behaviour is generally due to the programmer presuming a particular effect that is not matched by the choice made by the implementation. As indicated above, many such failures are indicated by compile-time error messages or run-time exceptions. However, there are cases where the implementation-defined behaviour might be silently misconstrued, such as if the implementation presumes Ada.Exceptions.Exception\_Information returns a string with a particular format, when in fact the implementation does not use the expected format. If a program is attempting to extract information from Exception\_Information for the purposes of logging propagated exceptions, then the log might end up with misleading or useless information if there is a mismatch between the programmer’s expectation and the actual implementation-defined format.

Many implementation-defined limits have associated constants declared in language-defined packages, generally **package** System. In particular, the maximum range of integers is given by System.Min\_Int .. System.Max\_Int, and other limits are indicated by constants such as System.Max\_Binary\_Modulus, System.Memory\_Size, System.Max\_Mantissa, and similar. Other implementation-defined limits are implicit in normal ‘First and ‘Last attributes of language-defined (sub) types, such as System.Priority'First and System.Priority'Last. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.

### 6.57.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.57.5 of TR 24772-1.
* Be aware of the contents of Annex M of ISO/IEC 8652 [15] and avoid implementation-defined behaviour whenever possible.
* Make use of the constants and subtype attributes provided in package System and elsewhere to avoid exceeding implementation-defined limits.
* Minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, declare your own numeric types to match your particular application needs.
* When there are implementation-defined formats for strings, such as Exception\_Information, localize any necessary processing in packages with implementation-specific variants.

## 6.58 Deprecated Language Features [MEM]

This vulnerability does not apply to SPARK, since this is a new language syntax for SPARK. SPARK 2005 and earlier was different in its approach and tools used, hence there are no backward compatibility issues.

## 6.59 Concurrency – Activation [CGA]

This vulnerability does not apply to Spark because SPARK’s concurrency is restricted to Ada’s Ravenscar Tasking Profile. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. Therefore all resources required for task activation are allocated before the main program begins, and failure in activation will result in exceptions in the main program.

## 6.60 Concurrency – Directed termination [CGT]

This vulnerability does not apply to SPARK because Spark’s concurrency is restricted to Ada’s Ravenscar Tasking Profile. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. In addition, the Ravenscar Tasking Profile prohibits the “abort” statement, and Ravenscar tasks never terminate, hence directed termination is not possible, the resources are not freed and there is no risk of claiming a terminated task’s resources.. Tasks may be effectively removed from consideration by reducing their priority to below that of the idle task, thereby preventing execution.

## 6.61 Concurrent Data Access [CGX]

### 6.61.1 Applicability to language

SPARK’s concurrency is restricted to Ada’s Ravenscar Tasking Profile. Under this profile, tasks communicate exclusively using shared data or using a very limited form of protected objects. In either case, the language and profile guarantee that all data access is effectively single threaded and corruption of shared data or of protected data will be avoided. In spite of these rules, non-atomic data can be accessed and sequences of protected calls can update protected state in ways that are unsafe.

### 6.61.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.61.5 of TR 24772-1.
* Use protected objects for shared data.
* Statically determine that no unprotected data is used directly by more than one task.
* When shared variables are used, employ model checking or equivalent methodologies to prove the absence of race conditions.
* Use **pragma** Atomic and **pragma** Atomic\_Components to ensure that all updates to objects and components happen atomically.
* Use **pragma** Volatile and **pragma** Volatile\_Components to ensure that all tasks see updates to the associated objects or array components in the same order.

## 6.62 Concurrency – Premature Termination [CGS]

This vulnerability does not apply to SPARK because Spark’s concurrency is restricted to Ada’s Ravenscar Tasking Profile. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. In addition, the Ravenscar Tasking Profile prohibits the “abort” statement, and Ravenscar tasks never terminate, hence premature termination is not possible, the resources are not freed and there is no risk of claiming a terminated task’s resources.. Tasks may be effectively removed from consideration by reducing their priority to below that of the idle task, thereby preventing execution.

## 6.63 Protocol Lock Errors [CGM]

### 6.63.1 Applicability to language

SPARK is open to the errors identified in this vulnerability but supports a number of features that aid mitigation – see guidance below.

### 6.63.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.63.5 of TR 24772-1.
* Make use of loosely coupled communication using protected objects.
* Stay within the constraints defined by the Ravenscar Tasking profile [15].
* Verify with static analysis that exceptions cannot be raised in protected calls.
* Guard against protocol failures by using timed communication, watchdog timers (programmed using timed events) and time-stamped data (using the clock facilities). Do not use unprotected shared data for synchronization between tasks

## 6.64 Uncontrolled Format String [SHL]

### This vulnerability does not apply to SPARK since the language does not contain subprograms that use format strings.

# 7. Language specific vulnerabilities for C

[TBD]

# 8. Implications for standardization

# Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2004

[2] ISO/IEC TR 10000‑1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241 (all parts), *International terminology standards*

[4] ISO/IEC 9899:2011, *Information technology* — *Programming languages —* C

[5] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

[6] ISO/IEC 30170:2012, *Information technology* — *Programming languages —* Ruby

[7] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

[8] ISO/IEC 1539-1:2010, *Information technology — Programming languages —* Fortran — Part 1: Base language

[9] ISO/IEC 8652:1995, *Information technology — Programming languages —* Ada

[10] ISO/IEC 14882:2011, *Information technology* — *Programming languages —* C++

[11] R. Seacord, *The CERT C Secure Coding Standard*. Boston,MA: Addison-Westley, 2008.

[12] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2012 (third edition)16F[[2]](#footnote-2).

[13] ISO/IEC TR24731–1, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*

[14] ISO/IEC TR 15942:2000, *Information technology — Programming languages — Guide for the use of the Ada programming language in high integrity systems*

[15] Joint Strike Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

[16] Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in critical systems*, June 2008

[17] ISO/IEC TR 24718: 2005, *Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems*

[18] L. Hatton, Safer C: developing software for high-integrity and safety-critical systems. McGraw-Hill 1995

[19] ISO/IEC 15291:1999, *Information technology — Programming languages — Ada Semantic Interface Specification (ASIS)*

[20] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

[21] IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

[22] ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

[23] J Barnes, High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[25] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[26] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

[27] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 <http://www.embedded.com/1999/9907/9907feat2.htm>

[28] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

[29] Lions, J. L. [ARIANE 5 Flight 501 Failure Report](http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report). Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[30] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See <http://www.cert.org/books/secure-coding> for news and errata.

[31] John David N. Dionisio. Type Checking. <http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf>

[32] MISRA Limited. "[MISRA C](http://www.misra.org.uk/): 2012 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, March 2013 (ISBN 978-1-906400-10-1 and 978-1-906400-11-8).

[33] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation, (<http://cwe.mitre.org/>)

[34] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[35] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

[36] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[37] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 <http://www.nsc.liu.se/wg25/book>

[38] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, <http://archive.gao.gov/t2pbat6/145960.pdf>

[39] Robert Skeel, *Roundoff Error Cripples Patriot Missile*, SIAM News, Volume 25, Number 4, July 1992, page 11, <http://www.siam.org/siamnews/general/patriot.htm>

[40] CERT. *CERT C++ Secure Coding Standard*.  [https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637%20) (2009).

[41] Holzmann, Garard J., Computer, vol. 39, no. 6, pp 95-97, Jun., 2006, *The Power of 10: Rules for Developing Safety-Critical Code*

[42] P. V. Bhansali, A systematic approach to identifying a safe subset for safety-critical software, ACM SIGSOFT Software Engineering Notes, v.28 n.4, July 2003

[43] Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992. Available from: <http://www.adaic.org/docs/95style/95style.pdf>

[44] Ghassan, A., & Alkadi, I. (2003). Application of a Revised DIT Metric to Redesign an OO Design. *Journal of Object Technology* , 127-134.

[45] Subramanian, S., Tsai, W.-T., & Rayadurgam, S. (1998). Design Constraint Violation Detection in Safety-Critical Systems. The 3rd IEEE International Symposium on High-Assurance Systems Engineering , 109 - 116.

[46] Lundqvist, K and Asplund, L., “*A Formal Model of a Run-Time Kernel for Ravenscar*”, The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

[47] ISO/IEC TS 17961, *Information technology – Programming languages, their environments and system software interfaces – C secure coding rules*

[48] GNU Project. GCC Bugs “Non-bugs” http://gcc.gnu.org/bugs.html#nonbugs\_c (2009).

# Index

LHS (left-hand side), 22

1. This case is somewhat specialized but is important, since enumerations are the one case where subranges turn *bad* on the user. [↑](#footnote-ref-1)
2. The first edition should not be used or quoted in this work. [↑](#footnote-ref-2)